Effective Random Testing of Concurrent Programs

Koushik Sen
EECS Department, UC Berkeley, CA, USA.
ksen@cs.berkeley.edu

ABSTRACT

Multithreaded concurrent programs often exhibit wrong behaviors due to unintended interferences among the concurrent threads. Such errors are often hard to find because they typically manifest under very specific thread schedules. Traditional testing, which pays no attention to thread schedules and non-deterministically exercises a few arbitrary schedules, often misses such bugs. Traditional model checking techniques, which try to systematically explore all thread schedules, give very high confidence in the correctness of the system, but, unfortunately, they suffer from the state explosion problem. Recently, dynamic partial order techniques have been proposed to alleviate the problem. However, such techniques fail for large programs because the state space remains large in spite of reduction. An inexpensive and a simple alternative approach is to perform random testing by choosing thread schedules at random. We show that such a naive approach often explores some states with very high probability compared to the others. We propose a random partial order sampling algorithm (or RAPOS) that partly removes this non-uniformity in sampling the state space. We empirically compare the proposed algorithm with the simple random testing algorithm and show that the former outperforms the latter.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Testing tools

Keywords

random testing, concurrent programs

1. INTRODUCTION

Multithreaded programs often exhibit wrong behaviors due to unintended interferences among the concurrent threads. Such concurrent errors–such as data races and deadlocks–are often difficult to find because they typically happen under very specific interleaving of the executing threads. A traditional method of testing concurrent programs is to repeatedly execute the program with the hope that different test executions will result in different interleavings. There are a few problems with this approach. First, testing being carried out in a particular environment often fails to come up with interleavings that can potentially happen in different environments, such as under different system loads. Second, testing depends on the underlying operating system or the virtual machine for thread scheduling—it does not try to explicitly control the thread schedules; therefore, testing often ends up executing the same interleaving many times. Despite these limitations, testing is an attractive technique for finding bugs in concurrent systems for several reasons: 1) testing is inexpensive compared to sophisticated techniques such as model checking and verification, 2) testing often scales to very large programs.

An alternative approach to testing is *model checking* [8, 9, 14, 16, 17, 32] or systematic search of the state space. Model checkers systematically control the thread scheduler to explore all possible behaviors of a program. Being systematic and exhaustive in nature, model checkers can often prove that a concurrent system satisfies its desired properties. Unfortunately, a key problem with model checking is that it does not scale with program size. This is because the number of possible interleavings of a concurrent program often grows exponentially with the length of execution. This is the well-known *state-explosion problem.*

Partial order reduction methods [13, 14, 25, 31] have been proposed to address the state-explosion problem. Such methods exploit the fact that a number of interleavings of a concurrent system are equivalent to each other because they correspond to the different execution orders of various non-interacting (or independent) instructions from concurrent threads. Different execution orders of non-interacting instructions from concurrent threads result in the same overall final state; therefore, if one execution order finds a defect, such as a deadlock, a data-race, or an assertion violation, then all equivalent execution orders will also detect the defect. Such equivalent interleavings are abstracted in terms of a *happens-before* relation which defines a partial order over all the instructions executed during an execution. Concurrent executions having the same *happens-before* relation are *equivalent.* A set of equivalent executions is abstractly called a Mazurkiewicz's trace [20], or simply *a partial order.* The goal of partial order reduction techniques is to explore at least one execution from each partial order and avoid exploring more than one execution from the same partial order. Traditionally, partial order techniques perform static program analysis to determine the interacting (or *dependent*) instructions. However, due to the limitations of static analysis in the presence of pointers and heap memory, static partial reduction usually fails to achieve significant amount of reduction for real-world programs. More recently, a dynamic technique for partial order reduction [11] has been proposed to remove this limitation.

Although partial order reduction helps model checkers to reduce the search space, they still cannot fully explore all possible behaviors of large concurrent programs with limited time and memory resources. Moreover, model checkers often use depth-first search or breadth-first search strategies to explore the state space. If the time budget of a model checker is restricted, such search strategies often result in a very *localized* search of the state-space.

An attractive alternative to prevent model checkers from getting stuck in a localized search is to perform randomized search. A simple randomized search strategy would do the following. It would pick a random thread to execute at every execution point where a potential thread switch can happen. This simple approach would naturally perform better than traditional testing because it *explicitly* tries to control the scheduling of the threads, rather than depending on the underlying operating system for scheduling. On the other hand, unlike model checkers, the simple randomized search strategy would explore wide variety of interleavings without getting stuck in a local search.

A fundamental problem with the randomized search strategy is that it samples all possible *non-equivalent* thread interleavings in a very non-uniform way—some partial orders are sampled more often than the others. This is because the number of linearizations of partial orders vary widely from one partial order to another. This is not a desirable situation because most concurrency related errors, such as deadlocks, data-races, and assertion violations, are robust with respect to partial orders—if one linearization of a partial order exhibits an error, then all linearizations will also exhibit the error. A natural question to ask is "Can we sample partial orders more uniformly?"

This paper proposes a novel algorithm to sample partial orders more uniformly than the simple randomized algorithm. Specifically, we propose a random partial order sampling algorithm (or RAPOS). Unlike the simple randomized algorithm, RAPOS does not pick a single random thread at every point in execution where a potential thread switch can happen. Instead, RAPOS picks a random set of threads at every execution point such that the next instructions from those threads are independent of each other; the next instructions from the set of random threads are then executed simultaneously without imposing any particular ordering on the instructions. The instructions in the set being independent of each other, any ordering of the instructions would result in the same partial order. Therefore, by avoiding the sampling of a random ordering of the independent instructions (i.e., by picking a deterministic ordering), we impose probability one on a single ordering and probability zero on other equivalent orderings. This is the key insight behind RAPOS.

It is worth mentioning that one cannot simply randomize a dynamic partial order reduction algorithm to uniformly sample partial orders. This is because a dynamic partial order reduction algorithm requires the knowledge of all the dependencies among various instructions. In a randomized setting, we cannot dynamically collect all the dependencies among the instructions as we do not perform an exhaustive search. Specifically, a dynamic partial order reduction algorithm computes a persistent set at every state at runtime and uses this set to determine which threads can be executed next from the state. The computation of a persistent set at a given state requires the exploration of all states that can be reached from the given state. Unfortunately, this requires an exhaustive search, which is not possible in a randomized setting.

We have implemented the simple randomized algorithm and RAPOS for concurrent Java programs in a tool called CalFuzzer. We compare both algorithms on a variety of concurrent Java programs including open libraries and closed applications. In the experiments, we demonstrate the following facts.

- RAPOS samples partial orders more uniformly than the simple randomized algorithm. This implies that defects, such as deadlocks, data-races, and assertion violations, that are invariant with respect to all linearizations of a partial order are sampled more uniformly.
- The number of partial orders sampled by RAPOS after a fixed number of executions is significantly greater than the same number for the simple randomized algorithm.
- The expected number of executions required to find a defect using RAPOS is significantly less than the simple randomized algorithm.

Our implementation is extensible—other stateless explicit scheduling strategies for Java threads, including full-fledged stateless model checkers, can be implemented in the CalFuzzer framework. We have made the tool publicly available.

The rest of this paper is organized as follows. In Section 2, we give an overview of RAPOS using a simple motivating example. We describe the details of the RAPOS algorithm in Section 3. In Section 4 and Section 5, we describe the implementation of RAPOS in the CalFuzzer framework and the results of our experiments, respectively. Related work is discussed in Section 6 followed by conclusion.

2. MOTIVATING EXAMPLE

We describe the simple randomized algorithm and the random partial order sampling algorithm using a simple example and show that RAPOS samples partial orders more uniformly than the simple randomized algorithm.

Consider the two-threaded program in Figure 1. The program has two shared variables `x` and `y`, each having an initial value 0. Thread 1 updates the variable `y` three times before assigning 4 to `x`. Thread 2 raises an error if `x` is equal to 4. The program can exhibit five interleavings as shown in Figure 2. Out of these five interleavings, the last interleaving violates the assertion.

In the program, the instruction `x = 4;` in Thread 1 and the instruction `if (x==4) assert(false);` in Thread 2 are dependent as they access the same shared variable `x` and at

```
Initially x = 0 and y = 0

Thread 1:                  Thread 2:

y = 1;                     if (x==4) assert(false);
y = 2;
y = 3;
x = 4;
```

Figure 1: Simple Example

```
if (x==4) assert(false); y=1; y=2; y=3; x=4;
y=1; if (x==4) assert(false); y=2; y=3; x=4;
y=1; y=2; if (x==4) assert(false); y=3; x=4;
y=1; y=2; y=3; if (x==4) assert(false); x=4;
y=1; y=2; y=3; x=4; if (x==4) assert(false);
```

Figure 2: Five Interleavings Exhibited by the Simple Example

least one of the accesses is a write. As such, the program has two distinct partial orders. The first four interleavings in Figure 2 are linearizations of one partial order and the last interleaving is the linearization of the other partial order.

The simple randomized algorithm at any state picks one of the enabled threads uniformly at random and executes its next instruction. For example, in the initial state, the algorithm picks Thread 1 with probability 0.5 and Thread 2 with probability 0.5. If Thread 1 is picked, then the algorithm executes `y = 1;` and again flips a coin to pick either Thread 1 or Thread 2. This is repeated until the program terminates. A simple computation shows that the simple randomized algorithm will sample the first interleaving with probability 0.5, the second interleaving with probability 0.25, the third interleaving with probability 0.125, the fourth interleaving with probability 0.0625, and the fifth interleaving with probability 0.03125. Therefore, one partial order will be sampled with probability 0.96875 and the other will be sampled with probability 0.03125. This also implies that the assertion violation will be discovered with probability 0.03125. Ideally, we would like to come up with an algorithm that would sample each partial order with probability 0.5 so that we can detect the assertion violation with probability 0.5. We will show, in Section 3.5, that RAPOS will sample each partial order more uniformly.

3. ALGORITHMS

In this section, we give a detailed description of the simple randomized algorithm and the RAPOS algorithm. We describe our algorithms using a simple abstract model of concurrent systems.

3.1 Background Definitions

We consider a concurrent system composed of a finite set of threads. Each thread executes a sequence of statements and communicates with other threads through shared objects. In a concurrent system, we assume that each thread terminates after the execution of a finite number of statements. At any point of execution, a concurrent system is in a *state*. Let S be the set of states that can be exhibited by a concurrent system starting from the initial state s_0. A concurrent system evolves by transitioning from one state

```
RandomExecutionSimple() {
  $s := s_0$;
  while (Enabled($s$) $\neq \emptyset$) {
    take a random $t$ out of Enabled($s$);
    $s :=$ Execute($s, t$);
  }

  if (Alive($s$) $\neq \emptyset$) {
    print "Deadlock Detected";
  }
}
```

Figure 3: Simple Randomized Testing Algorithm

to another state. Let T be the set of all transitions of a system. We say $s \xrightarrow{t} s'$ to denote that the execution of the transition t changes the state s to s'. A transition is always caused by the execution of a statement by a thread. The behavior of a concurrent system can be expressed in terms of a transition system represented by a tuple (S, Δ, s_0), where

- S is the set of states of the system,
- $\Delta \subseteq S \times S$ is a transition relation defined as

$$(s, s') \in \Delta \text{ iff } \exists t \in T\colon s \xrightarrow{t} s'$$

- $s_0 \in S$ is the initial state.

`Enabled`(s) denotes the set of transitions that are enabled in the state s. `Alive`(s) denotes the set of threads whose executions have not terminated in the state s. A state s is in *deadlock*, if the set of enabled transitions at s (i.e. `Enabled`(s)) is empty and the set of threads that are alive (i.e. `Alive`(s)) is non-empty.

3.2 Simple Randomized Algorithm

A simple randomized algorithm to test a concurrent system is described in Figure 3. Starting from the initial state s_0, this algorithm, at every state, randomly picks a transition enabled at the state and executes it. The algorithm terminates when the system reaches a state that has no enabled transition. At the termination, if there is at least one alive thread, then the algorithm reports a deadlock.

3.3 Background Definitions for Partial Order Based Techniques

In this section, we recall some definitions from [13] that are necessary to describe our random partial order sampling algorithm. Partial order reduction methods in model checking exploit the fact that a number of execution paths of a concurrent system are equivalent to each other because they correspond to different execution orders of the same noninteracting transitions. Exploring different execution orders of noninteracting transitions always result in the same final state. If two transitions do not interact with each other, then we call them *independent*. For example, a transition denoting the acquire of a lock l_1 by a thread p_1 is independent of a transition denoting the acquire of a lock l_2 by another thread p_2, if l_1 and l_2 are different locks. Two transitions are said to be *dependent*, if they are not independent. Transitions from the same thread are always dependent on each other. Similarly, the acquire of the same lock by two threads are dependent on each other.

The execution of a concurrent system can be represented by a sequence of transitions. Specifically, $\tau = t_1 t_2 \dots t_n$ is a *transition sequence* if there exists states $s_1, s_2, \dots, s_{n+1}$ such that s_1 is the initial state and

$$s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \dots \xrightarrow{t_n} s_{n+1}$$

Note that in a transition sequence, a transition $t \in T$ can appear several times. We say that $t_i = t_j$ in a transition sequence iff $i = j$.

In a transition sequence, if we swap adjacent transitions that are independent, then the overall behavior of the transition sequence does not change. For example, if a transition sequence results in a deadlock state (or an assertion violation in a thread), then any transition sequence that is obtained by swapping adjacent independent transitions will also result in a deadlock state (or an assertion violation in the thread.) Thus, by swapping adjacent independent transitions in a transition sequence, we get other transition sequences that are *equivalent* with respect to finding bugs. This notion of *equivalence* can be formalized in terms of a *happens-before* relation $\preceq$ defined as follows. The *happens-before* relation $\preceq$ for a transition sequence $\tau = t_1 t_2 \dots t_n$ is defined as the smallest relation such that

1. if t_i and t_j are dependent and $1 \leq i \leq j \leq n$, then $t_i \preceq t_j$, and

2. $\preceq$ is transitively closed.

Thus $\preceq$ is a partial order relation. All transition sequences that are linearizations of the same partial order are *equivalent.* We will simply call such an equivalence class a *partial order* (often called Mazurkiewicz's trace.)

3.4 RAPOS: the Random Partial Order Sampling Algorithm

Partial order reduction methods exploit the fact that exploring one linearization of each partial order is sufficient for detecting any deadlock or assertion violation. Therefore, traditional model checking algorithms incorporating partial order reduction, explore at least one linearization of each partial order and try to avoid the exploration of more than one linearization of the same partial order. Unfortunately, model checkers usually do not scale for programs having large number of partial orders. Moreover, many model checkers that try to perform depth-first search or breadth-first search often end up doing very localized search of the state space. Randomized state space search tries to remove this limitation by guaranteeing that any execution path of a concurrent system can be sampled with almost uniform probability. The simple randomized testing algorithm presented in Section 3.2 is an example of such a randomized algorithm.

However, the simple randomized testing algorithm presented in Section 3.2 can end up sampling some partial orders more often than the others. This is because the number of linearizations of partial orders vary widely from one partial order to another. Given this non-uniformity, the simple randomized algorithm will sample partial orders having more linearizations more frequently than the others. We experimentally demonstrate this fact in Section 5.3. Another non-desirable consequence of this non-uniformity in the distribution of partial orders is that the probability of finding bugs that are exposed by partial orders having small number of linearizations would be very low. Next, we present an algorithm that will alleviate this problem.

Our random partial order sampling algorithm tries to sample partial orders more uniformly than the simple randomized algorithm. The algorithm is shown in Figure 4. In the algorithm, we use two sets, namely *schedulable* and *scheduled*, to compute a random thread schedule at runtime. These two sets are computed dynamically. We next informally describe the characteristics of these two sets.

- [***schedulable***]: At any state s in an execution, the set *schedulable* is a subset of the set of enabled transitions at the state s. The set `Enabled`$(s)\setminus$ *schedulable* at a state s contains those transitions whose execution we want to delay until we have seen the execution of a dependent transition. Therefore, the set `Enabled`$(s)\setminus$ *schedulable* behaves like a sleep set [13].

- [***scheduled***]: The set *scheduled* is a random subset of *schedulable* such that the transitions in *scheduled* are mutually independent. The set contains those transitions that our algorithm *actually* schedules next. Since the transitions in the set *scheduled* are independent of each other, we get the same partial order if we execute the transitions in *scheduled* in any order.

RAPOS works as follows. Initially, RAPOS initializes the set *schedulable* to the set `Enabled`(s_0). Then RAPOS does the following in a loop until there is no enabled transition. RAPOS randomly chooses a non-empty subset of *schedulable* such that the transitions in the subset are mutually independent. We call this subset *scheduled.* The function `RandIndependentSubset` (described in Figure 5) is used to choose this subset. RAPOS then executes the transitions in the set *scheduled* concurrently, i.e., without imposing any random ordering on the execution. This way RAPOS samples one execution with probability one and samples the other equivalent executions corresponding to different orderings with probability zero. Each transition in the set `Enabled`$(s)\setminus$ *scheduled* did not get an opportunity to execute in this iteration because RAPOS wants to delay their execution until RAPOS has seen the execution of a dependent transition. After the execution of the transitions in the set *scheduled*, RAPOS determines the transitions in the set `Enabled`$(s)\setminus$ *scheduled* that can be waked up from the delay. All such transitions and the next enabled transitions of the threads that RAPOS just executed are put in the set *schedulable.* The remaining transitions in the set `Enabled`$(s)\setminus$ *schedulable*(s) continue to delay as in a sleep set. It may happen that the set *schedulable* is empty because all the transitions in the set *scheduled* have got disabled and no transition from the set `Enabled`$(s)\setminus$ *scheduled* has been waked up. In such a case, to avoid the algorithm from getting stuck, RAPOS randomly selects a transition from the set `Enabled`(s) and put it in the set *schedulable.* RAPOS continues the loop.

In the randomized algorithm, rather than picking a transition from the set of all enabled transitions uniformly at random, we randomly compute a subset of enabled transitions (i.e. the set *scheduled*) such that the transitions in the set are mutually independent. Subsequently, the transitions in this subset are executed simultaneously without imposing

```
RAPOS() {
  $s := s_0$;
  $schedulable$ := Enabled($s$);
  while (Enabled($s$) $\neq \emptyset$) {
    $scheduled$ := RandIndependentSubset($schedulable$);
    for each $t \in scheduled$
      $s$ := Execute($s, t$);
    $schedulable := \{t' \in$ Enabled($s$) $\mid \exists t \in scheduled$
                    such that $t$ and $t'$ are dependent $\}$;
    if ($schedulable = \emptyset$)
      add a random element from Enabled($s$) to $schedulable$
  }

  if (Alive($s$) $\neq \emptyset$) {
    print "Deadlock Detected";
  }
}
```

Figure 4: Random Partial Order Sampling Algorithm (RAPOS)

```
// returns a non-empty random subset of $S$ such that all
// the transitions in the subset are mutually independent
RandIndependentSubset($S$: Set) {
    let $t$ be a random element in $S$
        $TmpSet := \{t\}$;
    foreach $t$ in $S$
        if ($\forall t' \in TmpSet$. $t$ and $t'$ are independent)
            add $t$ to $TmpSet$ with probability 0.5;
    return $TmpSet$;
}
```

Figure 5: Random Subset Selection Algorithm

any particular ordering. *This enables us to impose probability one on a single ordering and probability zero on other equivalent orderings.* Moreover, the set Enabled(s)$\setminus$ *schedulable* delays some transitions. This enables us to impose zero probability on the execution of transitions that would not contribute in the computation of a new partial order. This in turn results in a more uniform distribution over the distinct partial orders than the simple randomized algorithms. We demonstrate this fact using a simple example in Section 3.5 and empirically validate the fact in Section 5.

The soundness of the random partial order sampling algorithm is given by the following theorem. The proof of this theorem gives further insight on the working of the RAPOS algorithm.

THEOREM 1. *RAPOS samples any feasible interleaving of a concurrent system with non-zero probability.*

PROOF. Consider an execution $\tau = t_1 t_2 \dots t_n$ and let $\preceq$ be the happens-before relation for τ. Let $T = \{t_1, t_2, \dots, t_n\}$. If $T' \subseteq T$, then the set of minimal elements of T', denoted by $g(T')$, is defined as the set $\{t \in T' \mid$ there exists no $t' \in T'$ such that $t \neq t'$ and $t' \preceq t\}$. Given a subset T' of T, we also define $f(T') = \{t \in T \mid$ there is a $t' \in T'$ such that $t \neq t'$ and $t' \preceq t\}$, i.e. $f(T')$ is the set of all elements in T that are strictly greater than any element in T'. Let $h = g \circ f$ and $T_i = h^i(g(T))$. The following two lemmas hold.

LEMMA 2. *In the sequence $T_0, T_1, T_2, \dots$ there is a finite $i \geq 0$ such that for all $j \geq i$, $T_j = \emptyset$.*

PROOF. It follows from the definition of f and g that for any $i \geq 0$, T_i and T_{i+1} are disjoint. It also follows from the definition of f that for any $i \geq 0$, any element in T_{i+1} is greater than any element in T_i. Since T_i and T_{i+1} are disjoint and any element in T_{i+1} is greater than any element in T_i, for any $i \geq 0$ and $j > i$, T_i and T_j are disjoint and any element in T_j is greater than any element in T_i. Since T is finite, $h(\emptyset) = \emptyset$, and T_i and T_j are disjoint, the lemma holds. □

LEMMA 3. *Let m be such that for all $0 \leq j \leq m$, $T_j \neq \emptyset$ and $T_{m+1} = \emptyset$. Then $T = \cup_{0 \leq j \leq m} T_j$.*

PROOF. Consider an element $t'_i \in T$. Then we can find a maximal sequence $t'_0, t'_1, \dots, t'_i$, such that each element in the sequence is a distinct element of T and $t'_0 \preceq t'_1 \preceq \dots \preceq t'_i$. Then $t'_0 \in T_0$, $t'_1 \in T_1$, ..., $t'_i \in T_i$. Therefore, the lemma holds. □

The preceding lemmas show that $\{T_0, T_1, \dots, T_m\}$ is a disjoint partition of T and all transitions in a given T_i are mutually independent. Moreover, the functions f and g being deterministic, such a partition is unique for a given partial order. In the $(i+1)^{\text{th}}$ iteration of the RAPOS algorithm, there is a non-zero probability that the set *scheduled* is equal to T_i. Therefore, there is a non-zero probability of sampling an execution that is equivalent to τ. □

Although, we can prove the soundness of the algorithm, we do not know how to mathematically show that the algorithm samples partial orders more uniformly. However, the above proof gives the following important insight about the RAPOS algorithm. Given an execution τ, the sequence of sets of transitions $T_0, T_1, T_2, \dots$ gives a canonical representation of the partial order corresponding to τ. If RAPOS never executes the following statement
if (*schedulable* $= \emptyset$)
add a random element from Enabled(s) to *schedulable*
in Figure 4, then RAPOS will only execute these canonical representations (i.e., RAPOS will only sample distinct partial orders.) However, this ideal criteria is never met for real programs and we do not get a perfect uniform distribution over the partial orders. In Section 5, we empirically show that RAPOS samples partial orders more uniformly.

3.5 RAPOS on the Motivating Example

Consider the example in Figure 1. In the first iteration, RAPOS will assign one of the sets { `y=1;` }, { `if (x==4) assert(false);` }, { `y=1;` , `if (x==4) assert(false);` } to *scheduled* with probability 0.25, 0.25, and 0.5, respectively. If the set { `if (x==4) assert(false);` } is assigned to *scheduled*, then in the subsequent iterations the sets $\{y = 1;\}, \{y = 2;\}, \{y = 3;\}, \{x = 4;\}$ will be assigned to *scheduled* with probability 1. Therefore, the first interleaving in the Figure 2 will be sampled with probability 0.25. Similarly, if the set { `y=1;` , `if (x==4) assert(false);` } is initially assigned to *scheduled*, then the first or second interleaving in the Figure 2 will get sampled with probability 0.5. If the set { `y=1;` } is initially assigned to *scheduled*, then the last interleaving in Figure 2 will get sampled with probability 0.25. Note that the third and the fourth interleavings will never be sampled. Therefore, the two partial orders of the program will be sampled with probability 0.25 and 0.75, respectively. Therefore, RAPOS will detect the assertion violation with probability 0.25. This probability is significantly higher than 0.03125, the probability with which the simple randomized algorithm will detect the assertion violation.

4. IMPLEMENTATION

We have implemented our randomized algorithms for Java in a prototype tool called CalFuzzer. The tool is publicly available at `http://srl.cs.berkeley.edu/~ksen/calfuzzer/`. The tool provides a general framework for controlling the scheduler of a Java virtual machine and we believe that this tool can be used to experiment with various other algorithms that require total control over the Java thread scheduler. Next we briefly describe the architecture of CalFuzzer.

CalFuzzer consists of two core components: an *instrumentor* and a scheduler library, called the *director*. The instrumentor uses the SOOT compiler framework [30] to instrument Java code. Currently, we do not instrument the standard library that comes with the JDK. This is because we use the same library to implement the *director*. As such CalFuzzer cannot perform a thread switch when a thread is executing a standard library function. The instrumentor replaces all synchronization operations, such as `monitorenter`, `monitorexit`, `wait`, `notify`, `join`, `yield`, `sleep`, with function calls provided by the director. The `synchronized` keyword is replaced from all synchronized methods by calls to functions defined by the director.

During the execution of an instrumented Java program, all synchronization operations performed by the program at runtime are delegated as function calls to the director. The director implements the function calls to provide all functionalities of a Java thread scheduler that remains the same across all tools. For example, the director explicitly maintains the state of each active thread (e.g. whether a thread is waiting to acquire a lock) and the state of each active lock (e.g. which thread holds a lock) and updates the states whenever a synchronization operation is performed by the code under test. The director associates a binary semaphore with every thread. These semaphores are used to control the execution of individual threads. The director ensures that at any point during the execution only one thread is executing. In order to schedule a thread at every state where a thread switch can occur, the director implements a function that returns the next thread to be scheduled. We call this the `scheduler` function. The director assumes that the user of the framework customizes the behavior of the `scheduler` function to implement specific scheduling policies. A default implementation of the `scheduler` function based on the simple randomized algorithm is provided by the director.

CalFuzzer also provides various useful functionalities such as recording the thread choices made by the scheduler at every state in an execution. This enables CalFuzzer to deterministically replay an execution. The *happens-before* relation $\preceq$ is also tracked by the director. The *happens-before* relation is also hashed at the end of an execution using the MD5 hashing algorithm to generate a unique identifier for each partial order. We use this hash value in our experiments to count the number of times our algorithm has sampled a distinct partial order.

We have customized the `scheduler` function to implement the RAPOS algorithm. The implementation of RAPOS *allows thread switch only before a synchronization operation.* In [21], it has been shown that it is sufficient to perform thread switches before synchronization operations, provided that the algorithm tracks all data races. This particular restriction on thread switch keeps our implementation fast by avoiding redundant thread switches.

5. EMPIRICAL EVALUATION

5.1 Benchmark Programs

We evaluated our RAPOS algorithm on a variety of multithreaded programs. The benchmark includes both closed programs and open libraries which require test drivers to close them. The closed programs include three benchmark programs from the Java Grande Benchmark suite (raytracer, moldyn, and montecarlo) and three multithreaded programs implementing different patterns of synchronization (philo, boundedbuffer, and pipeline.) The latter three programs are taken from the Java PathFinder [32] distribution.

The open programs consist of several synchronized Collection classes provided with Sun's JDK 1.4.2, such as `Vector`, `ArrayList`, `LinkedList`, `LinkedHashSet`, `HashSet`, and `TreeSet`. Most of these classes (except the `Vector` class) are not synchronized by default. The `java.util` package provides special functions `Collections.synchronizedList` and `Collections.synchronizedSet` to make the above classes synchronized. In order to close the Collection classes, we wrote a multithreaded test driver for each such class. A test driver starts by creating two empty objects of the class. The test driver also creates and starts a set of threads, where each thread executes different methods of either of the two objects concurrently. We created two objects because some of the methods, such as `containsAll`, takes as an argument an object of the same type. For such methods, we call the method on one object and pass the other object as an argument. An example of such a test driver for the `LinkedList` class is shown below.

```
public class MTListTest extends Thread {
    List al1, al2; int c;
    public MTListTest(List al1, List al2,int c) {
        this.al1 = al1;
        this.al2 = al2;
        this.c = c;
    }
    public void run() {
        SimpleObject o1 = new SimpleObject(MyRandom.nextInt(3));
        switch(c){
            case 0:  al1.add(o1); break;
            case 1:  al1.addAll(al2); break;
            case 2:  al1.clear(); break;
            case 3:  al1.contains(o1); break;
            case 4:  al1.containsAll(al2); break;
            case 5:  al1.remove(o1); break;
            default:  al1.removeAll(al2); break;
        }
    }
    public static void main(String args[]){
        List al1 = Collections.synchronizedList(new LinkedList());
        List al2 = Collections.synchronizedList(new LinkedList());
        (new MTListTest(al1,al2,0)).start();
        (new MTListTest(al2,al1,1)).start();
        (new MTListTest(al1,al2,2)).start();
        (new MTListTest(al2,al1,3)).start();
        (new MTListTest(al1,al2,4)).start();
        (new MTListTest(al2,al1,5)).start();
        (new MTListTest(al1,al2,6)).start();
    }
}
```

5.2 Experimental Setup

We ran each of the benchmark programs (except moldyn, raytracer, montecarlo, and philo) 300 times using the simple randomized algorithm and the RAPOS algorithm. The programs from the Java Grande Benchmark suite are computationally intensive and take longer time to run; therefore, we restricted the number of executions of these programs to 100. We executed the philo program 2000 times, because it is the least computationally intensive program. We seeded each execution randomly. The random numbers were generated using a *Mersenne Twister* pseudo-random generator class. We picked this class to make sure that our random numbers are of high quality. We ran the experiments on 2 GHz Core Duo Laptop with 2GB RAM.

We use our experiments two demonstrate the following three hypotheses:

- The number of partial orders sampled by RAPOS after a fixed number of executions is significantly greater than the same number for the simple randomized algorithm.
- The expected number of executions required to find a defect using RAPOS is significantly less than the simple randomized algorithm.
- Probability of sampling a partial order by RAPOS is more uniform than the simple randomized algorithm. This implies that defects, such as deadlocks, data-races, and assertion violations, that are invariant with respect to all linearizations of a partial order are sampled more uniformly.

5.3 Results

Let us fix the following terminology. An experiment is equivalent to running a benchmark program n times using either the simple randomized algorithm or the RAPOS algorithm. In each experiment, we compute the partial order of each program execution and cluster the executions into distinct partial orders such that each cluster contains only those executions that have the same partial order. For each cluster C, we compute $|C|/n$ (where $|C|$ is the number of executions present in the cluster C) and use it to denote the probability of sampling the partial order corresponding to the cluster C. In each experiment, we also compute the number of distinct clusters or partial orders.

Table 1 shows the value of n (see column 4) for each experiment and the number of distinct partial orders encountered in each experiment (see column 5 for the simple randomized algorithm and column 6 for the RAPOS algorithm.) The higher the number of partial orders, the better the algorithm. Column 2 lists the source lines of code of each program (including comments and blank lines.) For the classes in `java.util`, we do not report SLOC, because these classes have a complex inheritance structure which prevented us from counting the actual SLOC. The total number SLOC of the classes inside `java.util` is around 25K.

Table 1 illustrates that the number of distinct partial orders sampled by RAPOS is several (i.e. between 1.38–18.75) times greater than the number of partial orders sampled by the simple randomized algorithm. In case of montecarlo, a program from the Java Grande Benchmark suite, using RAPOS, we obtained 75 distinct partial orders in 100 executions, whereas, the simple randomized algorithm resulted in a total of 4 distinct partial orders. This implies that certain partial orders in the case of montecarlo have significantly more linearizations compared to the others; therefore, our simple randomized algorithm ended up sampling more executions from the same partial order. In some programs, such as philo and boundedbuffer, the partial orders are more uniformly distributed; therefore, RAPOS performed a little better than the simple randomized algorithm on these programs. In summary, we conclude from the results in Table 1 that on the benchmark programs, the number of partial orders sampled by RAPOS after a fixed number of executions is greater than the same number for the simple randomized algorithm.

In Figure 6, we compare the probability of sampling distinct partial orders in each experiment. Each horizontal bar in the figure plots the relative probability of sampling a partial order. Specifically, each vertical segment in a horizontal bar represents a distinct partial order and the area of the segment gives the probability of sampling the partial order. The bars on the left column plot data for the simple randomized algorithm and the bars on the right column plot data for the RAPOS algorithm.

The plots in Figure 6 show that the probability of sampling a partial order by RAPOS is more uniform than the simple randomized algorithm. For example, in the case of `HashSet`, the probability of sampling distinct partial orders using the simple randomized algorithm varies from 0.003 to 0.51. In contrast, for the same program, the probability of sampling distinct partial orders using the RAPOS varies from 0.003 to 0.017, which is more uniform.

Defect Detection. Three programs, namely `Vector`, `ArrayList`, and `LinkedList`, in our set of benchmarks can throw `ConcurrentModificationException` exception if their method `x.containsAll(y)` is called concurrently with a method that modifies the object `y`. This is because these `containsAll` methods are implemented by the superclass `AbstractCollection` and the implementation uses iterator in a thread-unsafe way. However, the exception is thrown under very specific thread schedules. In order to compare the defect detection capabilities of the simple randomized algorithm and the RAPOS algorithm, we computed the expected number of executions required to expose the uncaught `ConcurrentModificationException` in the three programs. The expected numbers are computed by taking an average over 25 experiments.

Figure 7 shows the summary of the results of our defect detection experiments. The results demonstrate that the RAPOS algorithm catches the `ConcurrentModificationException` faster than the simple randomized algorithm. For example, for the `Vector` class, RAPOS detects the exception in 129 executions on an average, whereas, the simple randomized algorithm detects the exception in 615 executions on an average. These results support our hypothesis that RAPOS finds defects faster than the simple randomized algorithm.

5.4 Threats to Validity

A key external threat to validity is that we do not know if our selected benchmarks are representative of the multithreaded Java programs found in practice. We tried to minimize this threat by picking programs from various domains such as data structures, high-performance computing, and examples of various synchronization mechanisms. We have only compared RAPOS with the simple randomized

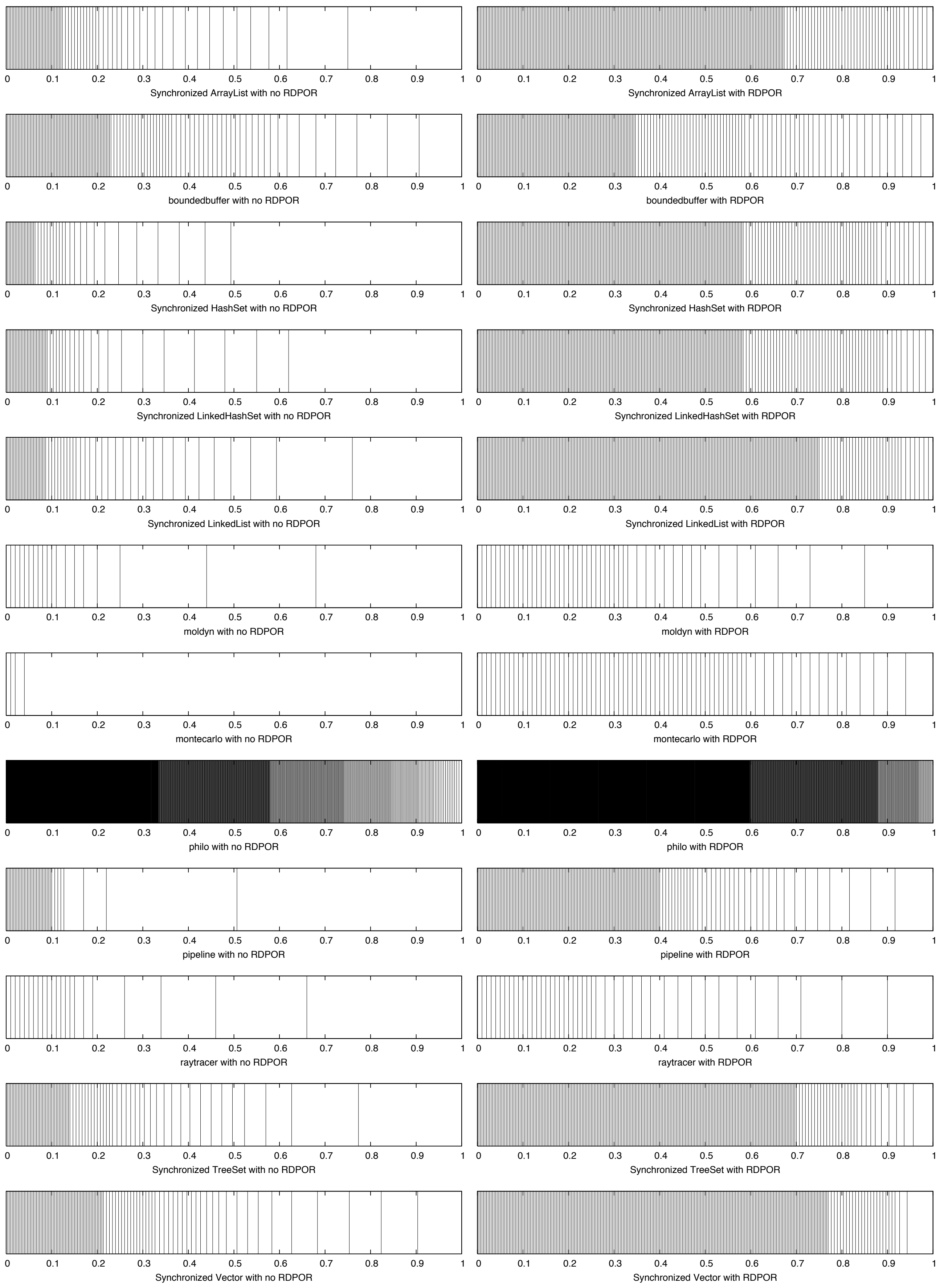

Figure 6: Relative Distribution of Partial Orders Sampled by the Simple Randomized Algorithm and the RAPOS algorithm

Program	SLOC	# of Threads	Total # of Executions	# of distinct Partial Orders		Column 6/Column 5
				Simple Algorithm	RAPOS	
ArrayList	-	7	300	71	247	3.47
boundedbuffer	141	4	300	118	171	1.45
HashSet	-	7	300	42	230	5.48
LinkedHashSet	-	7	300	48	230	4.79
LinkedList	-	7	300	58	259	4.47
moldyn	1291	11	100	19	48	2.53
montecarlo	3557	11	100	4	75	18.75
philo	91	3	2000	1124	1552	1.38
pipeline	119	8	300	38	156	4.11
raytracer	1859	11	100	22	44	2.00
TreeSet	-	7	300	78	240	3.08
Vector	-	7	300	105	256	2.44

Table 1: Number of Partial Orders Sampled after a Fixed Number of Executions

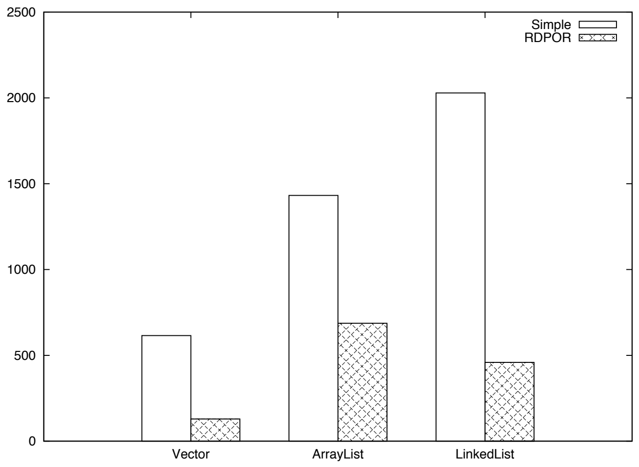

Figure 7: Expected Number of Executions Required to Detect a Defect

algorithm. This could be seen as another threat to validity. However, we believe that our simple randomized algorithm, being explicit in controlling thread schedules, will outperform existing random testing algorithms for concurrent programs [10, 29].

Internal threats to validity include our implementation bias that could affect our results. Defects in our tool can amplify such biases. To minimize this threat, we have extensively tested our tool on a number of small multithreaded programs. Moreover, in our simple randomized algorithm, we have made sure that we switch threads only at the synchronization points.

Our conclusion that RAPOS can find bugs more quickly than the simple randomized algorithm might not hold if bugs exist in partial orders that have comparatively large number of linearizations. In such cases, the simple randomized algorithm will be able catch those bugs faster than RAPOS. However, our limited experiments show that concurrency related bugs often exist in partial orders having comparatively smaller number of linearizations. This also matches our intuition because hard to find concurrency bugs often manifest under very specific and corner-case thread schedules.

6. RELATED WORK

Random testing [1, 4, 5, 12, 18, 19, 23, 24] (or fuzz testing) is widely used to test sequential programs. Random testing has several advantages: it is fast and it can find common bugs quickly [12]. Although random testing has been quite successful in finding bugs, the problem with such random testing is twofold: first, many sets of values may lead to the same observable behavior and are thus *redundant*, and second, the probability of selecting particular inputs that cause buggy behavior may be astronomically small [22].

In concurrent programs, if we fix the data inputs, then random testing can be used to pick a random thread for scheduling at every synchronization point. Since the number of threads executing concurrently at any time is often small, the domain of random choices is small compared to sequential programs with inputs. As such random testing performs better for concurrent programs. However, the challenge lies in controlling the scheduler.

Recently, a couple of random testing techniques [10, 29] for concurrent programs have been proposed. These techniques randomly seed a Java program under test with the `sleep()`, the `yield()`, and the `priority()` primitives at shared memory accesses and synchronization events. As a result the scheduling gets perturbed randomly at runtime. Although these techniques have successfully detected bugs in many programs, they have two limitations. These techniques are not systematic as the primitives `sleep()`, `yield()`, `priority()` can only advise the scheduler to make a thread switch, but cannot force a thread switch. Second, reproducibility cannot be guaranteed in such systems [29] unless there is builtin support for capture-and-replay [10]. The simple randomized algorithm given in Section 3.2 removes these limitations by explicitly controlling the scheduler.

Randomized algorithms for model checking have also been proposed. For example Monte Carlo Model Checking [15] uses random walk on the state space to give probabilistic guarantee of the validity of properties expressed in linear temporal logic. Statistical model checking techniques [27, 33] verify probabilistic models against probabilistic properties approximately within probabilistic error bounds. Randomized depth-first search and its parallel extension [7] have been developed to dramatically improve the cost-effectiveness of state-space search techniques using parallelism. This technique uses the `RandomOrderScheduler` of JPF [32].

Several systematic and exhaustive techniques [2, 3, 26, 28] for testing concurrent and parallel programs have been developed recently. These techniques exhaustively explore all interleavings of a concurrent program by systematically switching threads at synchronization points.

7. CONCLUSION

Random testing (or fuzz testing) is a simple, inexpensive, yet effective technique for finding bugs in programs. So far, a couple of techniques for random testing of concurrent programs has been proposed. Those techniques try to partially control the underlying scheduler in a random manner. We propose a systematic algorithm for random testing concurrent programs by extending ideas from the model checking literature. We empirically demonstrate the effectiveness of our algorithm.

Acknowledgements

We would like to thank Benjamin Hindman and Pallavi Joshi for providing valuable comments on a draft of this paper. This work is supported in part by the NSF Grant CNS-0720906.

8. REFERENCES

[1] D. Bird and C. Muñoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[2] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.

[3] R. H. Carver and Y. Lei. A general model for reachability testing of concurrent programs. In *6th International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 76–98, 2004.

[4] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *ICFP 00*, pages 268–279. ACM, 2000.

[5] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[6] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE 06: Automated Software Engineering*, pages 59–68. IEEE, 2006.

[7] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 3–12. IEEE, 2007.

[8] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25(2–3):199–240, 2004.

[9] J. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[10] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, , and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[11] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of the 32nd Symposium on Principles of Programming Languages (POPL'05)*, pages 110–121, 2005.

[12] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*, 2000.

[13] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer-Verlag, 1996.

[14] P. Godefroid. Model checking for programming languages using verisoft. In *24th Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[15] R. Grosu and S. A. Smolka. Monte carlo model checking. In *11th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *LNCS*, pages 271–286, 2005.

[16] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *Int. Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[17] G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[18] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *ISSRE 05*, pages 267–276, 2005.

[19] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE 07: International Conference on Software Engineering*. ACM, 2007.

[20] A. W. Mazurkiewicz. Trace theory. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986*, volume 255 of *LNCS*, pages 279–324. Springer, 1986.

[21] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *ACM Symposium on Programming Language Design and Implementation (PLDI'07)*, 2007. (To Appear).

[22] J. Offut and J. Hayes. A Semantic Model of Program Faults. In *ISSTA 96*, pages 195–200. ACM, 1996.

[23] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 05: European Conference Object-Oriented Programming*, LNCS 3586, pages 504–527, 2005.

[24] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. IEEE, 2007.

[25] D. Peled. All from one, one for all: on model checking using representatives. In *5th Conference on Computer Aided Verification*, pages 409–423, 1993.

[26] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa verification conference 2006 (HVC'06)*, Lecture Notes in Computer Science. Springer, 2006.

[27] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 202–215, 2004.

[28] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 157–168. ACM Press, 2006.

[29] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Workshop on Runtime Verification (RV'02)*, volume 70 of *ENTCS*, 2002.

[30] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON 1999*, pages 125–135, 1999.

[31] A. Valmari. Stubborn sets for reduced state space generation. In *10th Conference on Applications and Theory of Petri Nets*, pages 491–515, 1991.

[32] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th International Conference on Automated Software Engineering*. IEEE Computer Science Press, Sept. 2000.

[33] H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 223–235. Springer, 2002.